

一、本周工作

阅读《C++ concurrency in action》Chapter 5. The C++ memory model and operations on atomic types（内容提炼见附录）。

二、下周工作

1. 秋学期考试
2. 阅读Chapter 6. Designing lock-based concurrent data structures.

附录

Chapter 5. The C++ memory model and operations on atomic types

本章包含的内容有：

- C++11内存模型
- C++11标准库提供的原子类型
- 原子类型支持的操作
- 如何使用这些操作为线程提供同步

这些内存模型、原子类型及其操作，通常会被精简为1、2条CPU指令，提供了低层次的并行实现。

它们也非常复杂，除非需要利用这些原子类型编写同步代码，否则并不需要了解这些细节。

“

This is quite complex: unless you're planning on writing code that uses the atomic operations for synchronization (such as the lock-free data structures in chapter 7), you won't need to know these details.

5.1 Memory model basic

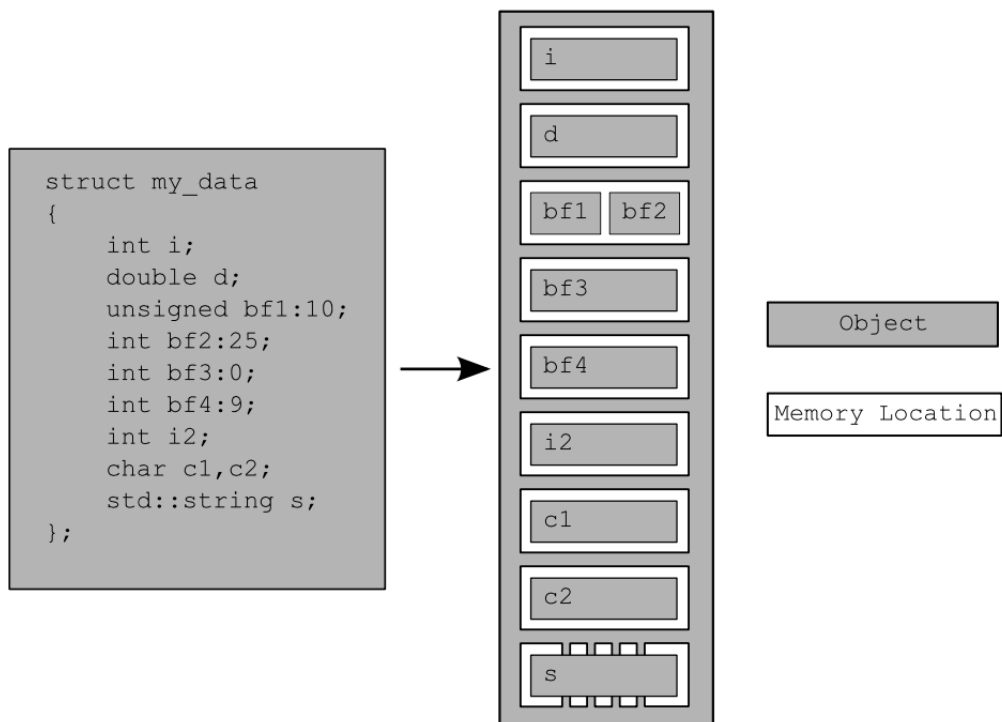
内存模型由两部分组成：

- 结构部分(structural), 物体(things)如何在内存中布局
- 同步部分(concurrency)

5.1.1 Objects and memory locations

C++程序中的所有数据都由**对象**组成。这个对象并不是我们通常OO(Object Oriented)所描述的对象，这种对象有如下特点：

- C++标准将对象定义为“一块存储区域” (“a regin of storage”)
- 对象可以是基础类型，如int, double等；也可以是用户定义的类型
- 对象中可以包含子对象(subobject)
- 对象存储在一个或多个内存位置（memory locations）中
- 运用位域（bit fields）时，它们被看做不同的对象，但仍使用同一块内存位置



从上图可以得出四点信息：

- 每一个变量，都是一个对象
- 每个对象至少占据了一块内存位置
- 基础类型，如int或char，都占据了有且只有一块内存地址，即使它们为相邻变量
- 相邻位域占据同一块内存地址（另，主要到bf3把bf4分开，让它占据了一块自己的内容地址）

5.1.2 Objects, memory locations, and concurrency

强调了若两个线程几乎同时访问一块内存位置，并且其中至少有一个访问是写操作，其中至少有一个操作不是原子性的，这种数据竞态将导致未定义的行为。

5.1.3 Modification orders

5.2 Atomic operations and types in C++

5.2.1 The standard atomic type

简述了库中的原子类型：

- `atomic_flag`
- `atomic<>`模板

5.2.2 Operations on `std::atomic_flag`

`atomic_flag`用来表示布尔类型。它的初始化方式比较特别：

```
std::atomic_flag f = ATOMIC_FLAG_INIT;
```

它是所有原子类型中，唯一一个要求这种初始化方式的类型，同时它也是唯一保证肯定是被无锁（lock-free）地实现的类型。

`atomic_flag`对象支持的操作只有：

- `destroy`, 析构函数
- `clear`, `clear()`成员函数
- `set`, `test_and_set()`成员函数

本小节同时给出了通过`atomic_flag`实现的自旋锁（spinlock）

```
#include <atomic>

class spinlock_mutex
{
public:
    spinlock_mutex(): flag(ATOMIC_FLAG_INIT) {}

    void lock()
    {
        while (flag.test_and_set()) {}
    }

    void unlock()
    {
        flag.clear();
    }

private:
    std::atomic_flag flag;
};
```

[点击此处](#)，参阅test_and_set()。

atomic_flag的局限性很强，由于不支持不修改状态的查询操作（nonmodifying query operation），甚至无法作为通用的布尔类型使用。因此，我们下面介绍atomic类型。

5.2.3 Operations on std::atomic<bool>

atomic<>和atomic_flag一样，没有拷贝构造和移动构造函数，成员函数包括且不仅限于：

- store
- load
- exchange

atomic可能不是无锁的，可以通过is_lock_free()成员函数来检测。

STORING A NEW VALUE (OR NOT) DEPENDING ON THE CURRENT VALUE

```
atomic.compare_exchange_weak(expected, value);
```

上述函数的语义为：

- 若atomic的值与expected的值相同，那么atomic = value, 返回true
- 反之，expected = value, 返回false

但是，对于那些不含compare-and-exchange指令的CPU而言，这个函数将被翻译成多条CPU

指令，若这些指令在执行的过程中，线程被调度换出，那么这个函数将执行失败，这种情况被称为spurious failure. 为应对这种情况，上述代码可改为：

```
bool expected = false;
extern atomic<bool> b;
while (! b.compare_exchange_weak(expteced, true) && !expected) {}
```

5.2.4 Operations on std::atomic<T*>: pointer arithmetic

指针及下述类型的操作与atomic相似，暂不描述。

5.2.5 Operations on standard atomic integral types

5.2.6 The std::atomic<> primary class templates

5.2.7 Free functions for atoimc operations

5.3 Synchronizing operations and enforcing ordering

给出了一个先写后读的示例，用来引出happen-before和synchronized-with关系。

5.3.1 The synchronizes-with relationship

只有对原子类型的操作，才能引入synchronizes-with关系。

可以暂时假想，线程A写(store)了原子数据x，线程B读(store)了原子数据x，那么AB间存在synchronizes-with关系。

5.3.2 The happens-before relationship

happens-before

对于单线程程序，如果操作A在操作B之前（sequence before），那么A also happens-before B.

一般来说，在一个statement内发生的operations中没有happens-before关系，因为标准未对其进行定义。一些特例是，逗号表达式或一些A的输出被作为B的输入的statement.

当然，一个statement内的所有操作As，都对其接下来statement内的所有操作Bs，都具有happens-before关系。

inter-thread happens-before

对于多线程程序，如果线程P1中的操作A ***inter-thread happens-before*** 线程P2中的操作B，那么A happens-before B.

我认为，inter-thread happens-before的定义为：若操作A synchronizes-with 操作 B, 那么A inter-thread happens-before B.

inter-thread happens-before与sequenced-before有如下关系：

- 若A is sequenced before B, B inter-thread happens-before C, 那么A inter-thread happens-before C.

inter-thread happens-before具有传递性。

5.3.3 Memory ordering of atomic operations

...

介绍了所有的内存序和常见组合，详情可参阅[链接](#)。